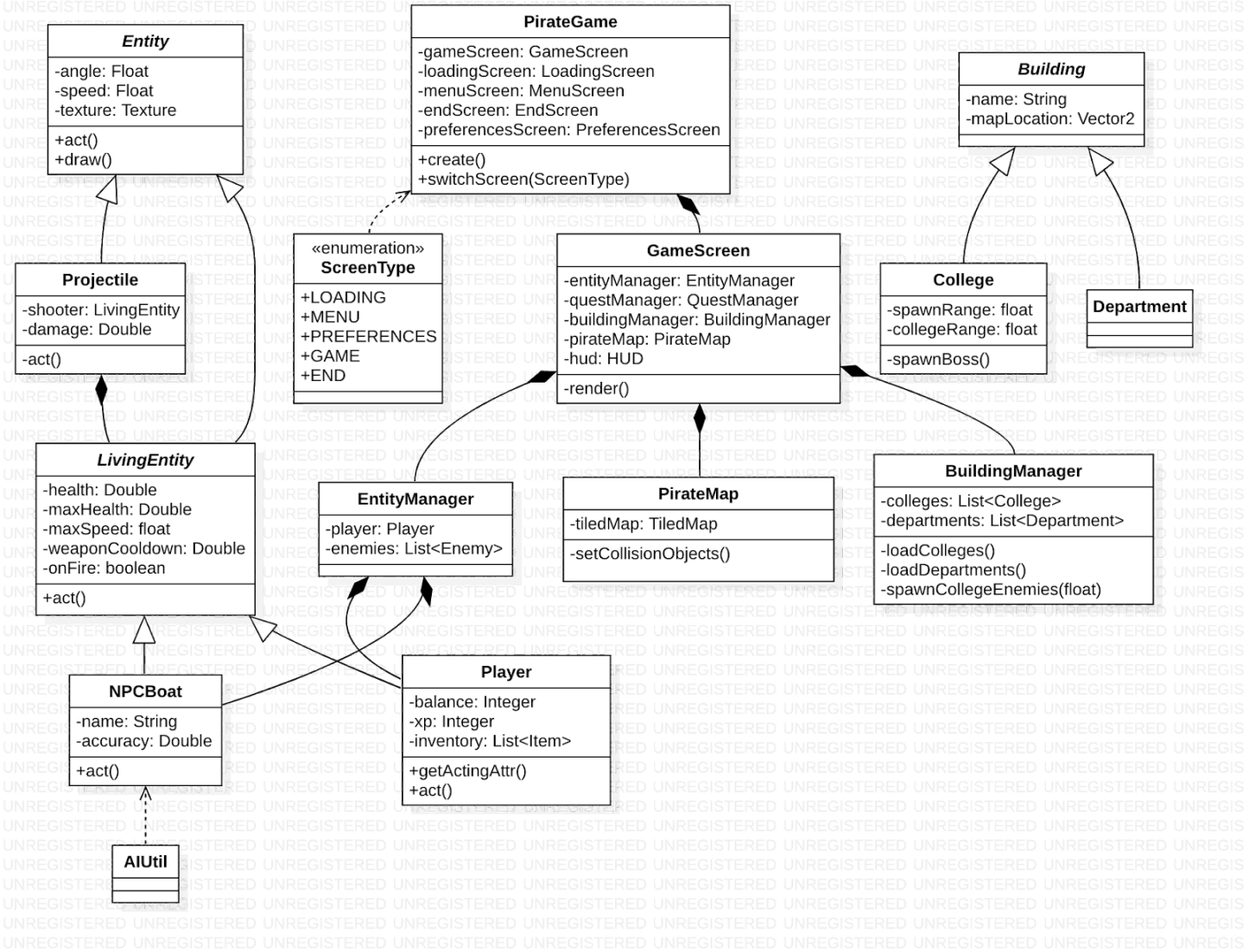
Architecture

Stakeholders: Richard Paige, University of York Communications Office
Team: Barney Morgan, Cameron Smith, Harry Berge, Jake Phillips, Matthew Wilkie, Rob Weddell

Concrete System Architecture

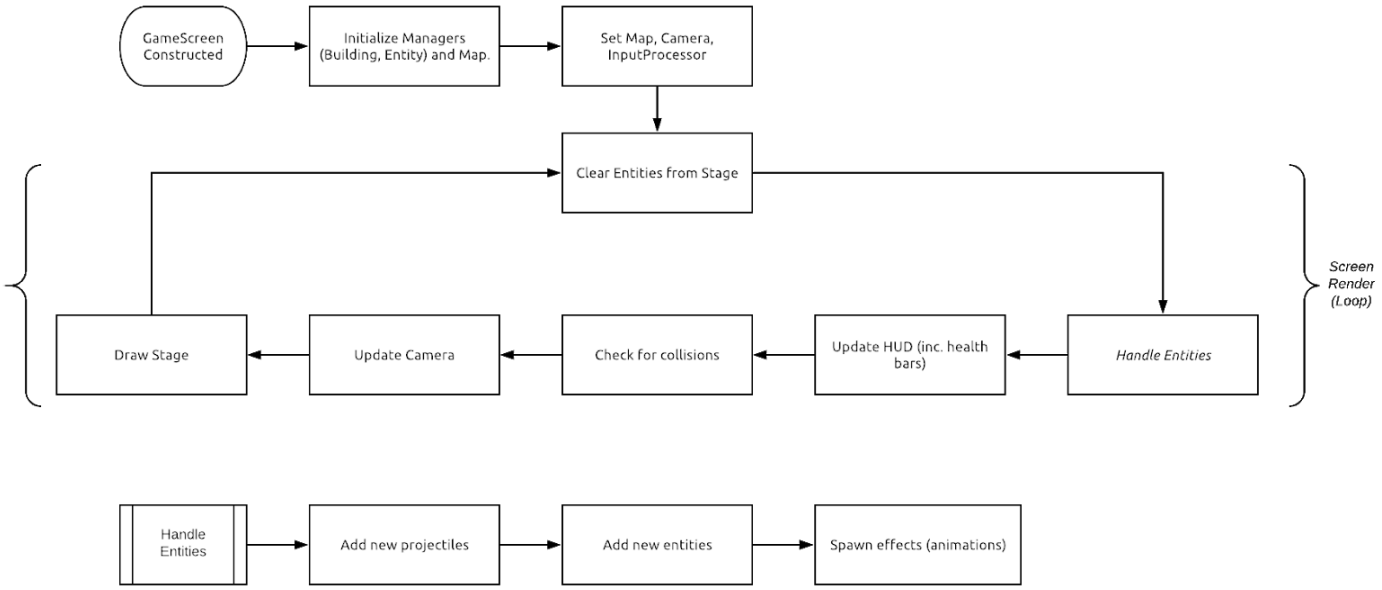
Following our proposed system architecture, we were able to develop our game accordingly. We used all tools as initially discussed including; Java as our language of choice, LibGDX [1] as our game engine, Gradle [2] as our dependency/build manager and TravisCI [3] for continuous integration (build tests).

The main changes to our initial plan came as a result of actually implementing the game's mechanics and reevaluating how best to create them including the addition of new ideas. In the case of one feature initially mentioned, multiple projectile types, we decided to remove this due to time constraints (risk 4). See below for class diagram [4].



The tool used for the diagram was StarUML [5] produced in UML 2.0 [6]. We have omitted details from the diagram related to questing which was not entirely implemented and largely unchanged from the initial proposal.

Similar changes to our initial architecture can be seen in the flow chart [7] below, produced with LucidCharts [8]. Other, previously unconsidered, checks are required in the main loop of the game: for example, collisions (between the player, NPCs, projectiles and map objects).



As before, this flow chart is simply representational of the behaviour of the game and not how it is implemented (event-driven architecture).

In attempts to make our game easily maintainable and understandable by others, we have opted for many modular functions. For example, instead of hard-coding instances of Colleges, Departments, Items and (in the future) quests, we have loaded these from a simple JSON file - allowing them to be edited easily. In regards to the buildings (Colleges and Departments), the JSON file entries include a value which associates the instance with an object loaded from the TiledMap, so they appear in the correct position. The JSON file also includes details such as enemy difficulty and spawn range.

Furthermore, we have created a new class called NPCBuilder which is used to easily create custom enemies and even generate randomized enemies to add variety to the game (req 3.1).

Concrete System Justification

The class diagram [4] builds heavily upon our abstract diagram but now wholly reflects our implementation and removes any ambiguity from the abstract design. Thus, below is the breakdown of the classes along with what requirements [9] they relate to.

The **Building** objects handle the departments and colleges that can be interacted with in-game, relating to requirement **2.1**.

Building: Abstract base class for **College** and **Department**. Has shared fields.
BuildingManager: Loads colleges and departments from JSON file into memory. Also responsible for setting their position on the map and spawning NPCBoats at college locations.
College: Extends Building and acts as a spawn point for enemies until the player has defeated the boss and allied the college. Has a varying difficulty as the game develops, getting harder the further the player gets into the game.
Department: Extends Building and will allow the player to fix their ship in exchange for in-game gold (not yet implemented). Relates to requirement **2.13**.

The **Entity** object handles the moving effects and animations that appear on the game screen as well as the in-game mechanics.

AnimationManager: Manages the effects and animations in the game screen.
Entity: Base class which draws the entities on the screen during act call.
EntityManager: Responsible for the management of the Player and all NPCBoat instances. Has utility methods for retrieving certain entities (at a location, in range, etc..)
LivingEntity: Abstract class extending Entity. Adds fields related to living (damageable) entities including health, weapon cooldown and max speed. Is also responsible for changing speed during act call.
NPCBoat: Gives the NPC boat life-like characteristics (AI). Allowing the NPC boat to rotate based on the nearest target and dodges projectiles directed at them. Relates to requirement **2.6**.
NPCBuilder: Provides easy interface to build/customize NPCs including generating randomized NPCs based on difficulty value. Relates to requirement **2.8**.
Player: Deals with the player inputs via the use of keyboard and mouse, allowing the player to accelerate/decelerate and turn the boat and fire projectiles at enemies. Relates to requirement **2.2, 2.6, 2.18**.

The **Item** objects handle the items that can be found and used in-game, relating to the requirement **2.16**.

Item: This class sets the lore for each item, producing the attributes for each individual Item.
ItemManager: Loads in Items from JSON file and deals with the use of the items, whether the item is equipped or consumed.
Reward: Manages the rewards gained throughout the playing of the game. This includes the gold, xp and items, relating to the requirement **2.11**.

The **Projectile** object handles the projectiles across the map. These vary with the weapon used and have a distance limit from the shooter to adhere to. The projectiles provides the player with a means to combat, relating the the requirements **2.3, 2.6, 2.12**.

Projectile: Extension of Entity, this class deals with the combat in the game. It tracks the fired projectiles and applies the appropriate damage for the respective projectile.
ProjectileManager: Loads in the projectiles from JSON via DAO and spawns in the new projectile when it is cast, adding it to the list of active projectiles on the map. Also, it removes non-active projectiles that are too far away from the shooter.

PirateMap: Initialises the game by loading in the Tiled map in addition to as setting the spawn point of the player and producing a collision layer to prevent ship sailing onto land. Relates to the requirement **2.15**.

The **Screen** object dictates which screen the user interacts with. The Screen object has the following classes:

GameScreen: Loads up the game screen and starts the game. Switches to Menu Screen if the player dies in the game.

MenuScreen: Loads up the menu screen and switches to Game Screen the user selects to start the game.

ScreenType: Holds the options of screen type, these are: Menu, Preferences, Game, and End Screen.

HealthBar: Adds a health bar to Living Entities which updates and shows the health of the Living Entities.

HUD: Adds other information to the Game Screen. Gold accumulation is displayed on the screen so that the player can see their gold count and use it appropriately as well as the area of the map the player is in.

The **Utils** object handles the AI of the NPC boats, adding difficulty to the game which relates to the requirement **2.6, 2.19**. It has the following class:

AIUtil: Provides the mechanics for the AI entities for the player to battle. Calculating the perfect shot to hit the player, which is then used in conjunction with the difficulty of the NPC boat to produce a projectile path appropriate for the difficulty. The operation of our functions have documented [10].

ShapeUtil: Adds methods relating to shape checks, including checking whether certain shapes overlap.

PirateGame: Initialises the screen. Displays the menu screen or game screen, depending on which one the player has selected.

TextureManager: Handles the in-game textures. This includes the player, enemy, cannon balls, enemy dying, explosions.

We have omitted classes above which we have not developed due to the scope of assessment 2 (e.g. Quests)

<u>Changes from abstract architecture</u>

Our initial abstract architecture was built upon heavily with a few changes to fix any impracticalities in the abstract architecture as well as removing any redundant features and adding anything that was previously unconsidered. This includes our previously defined ‘Attribute’ class which we decided should be combined with ‘LivingEntity’ because the variables of an Attribute could easily be represented directly on a LivingEntity. The proposed classes ‘Boost’, ‘Ability’, ‘ConsumableItem’ and ‘EquipItem’ were removed and their proposed features moved into other classes as required. Possibly the biggest change to our abstract plan was the addition of the ‘AnimationManager’ and the change of ‘Enemy’ to ‘NPCBoat’. Firstly, AnimationManager was created to house our various forms of animation (water trail, death) which help to add immersion and finish to the game (req 3.2). Moving it to a single class allowed us to abstract away from single-use implementations for each type of Entity. The change to NPCBoat was as a result of the desire to have friendly boats in our game (req 2.8) - the name Enemy is, therefore, not befitting. Allied colleges now spawn NPCBoats which are friendly towards the player.

References

[1] LibGDX Website [Online]. Available: https://libgdx.info/ [Accessed 5 Oct. 2018].
[2] Gradle Website [Online]. Available: https://gradle.org/ [Accessed 12 Oct. 2018].
[3] TravisCI Website [Online]. Available: https://travis-ci.org/ [Accessed 3 Nov. 2018].
[4] Element of SEPRise!, UML Class Diagram [Online]. Available: https://sepr4.github.io/web/submission/assessment2/uml/ClassDiagram.pdf [Accessed 15 Jan. 2019]
[5] StarUML Website [Online]. Available: http://staruml.io/ [Accessed 19 Oct. 2018].
[6] UML Website [Online]. Available: http://www.uml.org/ [Accessed 12 Oct. 2018].
[7] Element of SEPRise!, Flowchart [Online]. Available: https://sepr4.github.io/web/submission/assessment2/flowchart/GameScreen.png [Accessed 15 Jan. 2019]
[8] LucidCharts Website [Online]. Available: https://www.lucidchart.com [Accessed 20 Oct. 2018].
[9] Element of SEPRise!, Updated Requirements [Online]. Available: https://sepr4.github.io/web/submission/assessment2/updated/Req2.pdf [Accessed 1 Jan. 2019]
[10] Element of SEPRise!, NPCBoat AI Documenation [Online]. Available: https://sepr4.github.io/web/submission/assessment2/NPCAI.zip [Accessed 1 Jan. 2019]